Memory Fault Simulator for Static-Linked Faults

Memory Fault Simulator for Static-Linked Faults

A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, P. Prinetto
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
E-mail {benso, bosio, dicarlo, dinatale, prinetto}@polito.it
http://www.testgroup.polito.it

Abstract

Static Linked Faults are considered an interesting class of memory faults. Their capability of influencing the behavior of other faults causes the hiding of the fault effect and makes test algorithm design and validation a very complex task. This paper presents a Memory Fault Simulator architecture targeting the full set of linked faults.

1. Introduction

Memories are one of the most important components in digital systems, and semiconductor memories are nowadays one of the fastest growing technologies. System-On-a-Chip (SOC) technologies allow to embed in a single chip all the components and functions that historically were placed on a hardware board. Within SOCs, embedded memories are the densest components, accounting for up to 90% of chips area [1]. It is thus common finding, on a single chip, tens of memories of different types, sizes, access protocols and timing. Moreover they can recursively be embedded in embedded cores.

The main issue in memory test is to define comprehensive fault models able to carefully represent the most common defects occurring in the production phase of the chips. On the other hand once a new fault models has been defined a memory test algorithm, able to detect it, must be developed [2] [3] [4] [5] and obviously validated. Memory fault simulation is therefore necessary to compute the Fault Coverage of a test sequence every time a new defect is discovered and the corresponding fault model defined.

An important class of memory faults is the class of linked faults [6]. A linked fault is a memory fault composed of two or more simple faults. The remaining ones can influence the behaviour of each simple fault and in some cases the fault can be masked.

Due to the complexity of both the fault models and the memory architecture, manual analysis [7] of the memory fault coverage is not anymore possible. In [8], a memory simulator (Memory Animation Package Plus, MAP+) has been proposed. This tool, developed at the Delft University of Technology, has been employed as a simulation tool for the evaluation of new and known test algorithms in presence of different faults (such as stuck-at, transition, coupling, address decoder, neighborhood pattern sensitive, read disturb faults, etc.). Although very interesting especially from an academic point of view, this tool does not allow a very detailed fault simulation.

[9] addresses only the fault coverage computation, without consider other characteristics such as power consumption

In our previous work [10] an architecture for a new flexible memory fault simulator, designed to address all the most critical issues in memories test generation and validation has been presented, in order to support the test engineer in optimizing the test algorithm and in addressing power consumption constraints. The tool is in fact able to compute the power consumption generated by the test input sequence, and to suggest a modification of the test algorithm in case its application does not fulfill a user-defined power consumption constraint.

To out best knowledge none of the previous works are able to deal with the complex class of linked faults. This paper presents a fault simulator architecture based on the architecture showed in [10] but extending it on the linked faults class. In particular we extend the fault model formalism and the simulator algorithm in order to be able to model the linked faults.

The paper is organized as follows. Section 2 introduces the overall tool architecture; Sections 3, 4 and 5 describe the memory and fault model representation used by the simulator. The fault simulation algorithm is described in

15th Asian Test Symposium (ATS'06)
0-7695-2628-4/06 $20.00 © 2006 IEEE

Section 6, whereas Section 7 gives some experimental results. Conclusions are summarized in Section 8.

2. The Fault Simulator Architecture

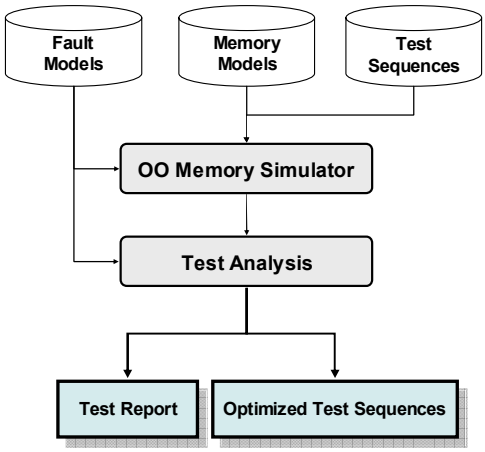

Figure 1. Simulator architecture

Figure 1 sketches the simulator overall architecture detailed in [10]. The *Object Oriented Memory Simulator* reads two main input files containing the memory functional and electrical models and the input test sequences, and simulates the execution of the input test sequence storing, for each memory cell, its logical and electrical temporal evolution. After the simulation, a Test Analysis module reads the target Fault Model files and computes their coverage w.r.t. the input test sequence. It generates two output files storing a detailed test report, and, whenever possible, an optimized input test sequence able to provide the same results of the original one.

The memory model is split in two parts:

- a *functional model*, represented as an Finite State Machine (see Section 4);
- an *electrical and physical model*, storing all the operating, technological, and topological characteristics of the memory;

The Fault Model files, formalized as collections of Fault Primitives (see Section 3), describe all the faulty behavior that the input test sequence is designed to detect. The Test Sequence files describe the sequence of operations applied to test the memory array. Using a proprietary language, it is possible to describe complex test algorithms as well as simple sequences of input patterns.

The Test Report file contains detailed information about:

- The Fault Coverage of each fault model and, when necessary, diagnostic information about the cells where the fault is not covered;
- The total power consumption caused by the application of the test sequences;

Finally, the Test Analysis module outputs an optimized test sequence, where redundant elementary operations not affecting the final fault coverage are removed. In the following sections we will detail the *functional model* that we modify in order to deal with linked faults, focusing in particular on the memory and fault models.

3. Fault Modeling

A *Functional Fault Model* (FFM) is a deviation of the memory behavior from the expected one under a set of performed operations. A FFM involves one or more *Faulty Memory Cells* (*f*-cells) classified in two categories: *Aggressor cells* (*a*-cells), i.e., the memory cells that sensitize a given FFM and *Victim cells* (*v-cells*), i.e., the memory cells that show the effect of a FFM. Each faulty behavior is sensitized by a sequence of *stimuli* applied on the *f*-cells. When dealing with RAMs, the applied stimuli are the memory operations. First of all we have to specify the *initial conditions* of the cell, i.e. the value (state) of the memory cell, where we are going to apply the operations. Hereinafter we resort to n as the size of the memory (i.e., the number of memory cells)

Definition 1: C is the set of the memory states (values), formalized as

$$C = \{0^{[i]}, 1^{[i]}, -^{[i]} \mid 0 \le i \le n\text{-}1\} \quad (1)$$

where the apex identifies the address of the cell. If the address is omitted, it means that the state can be applied on every memory cell indifferently. The '-' denotes a *don't care* condition.

Definition 2: X is the set of the memory operations, formalized as

$$X = \{r^{[i]}_{[d]}, w^{[i]}_{d} \mid 0 \le i \le n\text{-}1; d \in (0,1)\} \cup \{t\} \quad (2)$$

where:

- w^{i}_{d} is a *write* operation of the value d performed in the cell i;
- r^{i}_{d} is a *read* operation performed in the cell i. The value d is not strictly needed in case of a read operation. If used, it means the expected value that should be read from the i-th memory cell;
- t is a *wait* operation for a defined period of time. This additional element is needed to deal with *Data Retention Faults* [4].

If the address is omitted, it means that the operation can be applied on every memory cell indifferently. Each FFM can be described by a set of Fault Primitives (FPs) [14].

0-7695-2628-4/06 $20.00

Definition 3: A *Fault Primitive* FP represents the difference between an expected (fault-free) and the observed (faulty) memory behavior denoted by:

$$< S_a ; S_v / F / R > \quad (3)$$

Where S_a and S_v are the *Sequence of Sensitizing Operations and/or Conditions* respectively applied to *a*-cell and *v*-cell, needed to sensitize the given fault. The *j*-th condition/operation is represented as $c[x]$, where $c \in C$ (1), and $x \in X$ (2). $R = \{ (r)^n \mid r \in C \}$ is the sequence of values read on the aggressor cell when applying *S*.

As an example FP = $< 0w_1 ; 0 / 1 / - >$ means that the operation 'w_1' performed on the *a*-cell, when the initial state is 0 for both a and v cells, causes the *v*-cell to flip. No addresses are specified; therefore this fault can affect each couple of memory cell. Several FPs classification rules can be adopted, based on the number of memory operations (*m*) needed to sensitize the FP (static when *m* = 1 or dynamic fault elsewhere); and based on the number of memory cells (#FC) involved by the FP (single-cell where #FC = 1 or n-cells elsewhere fault) [14]. Since the FP notation not necessarily explicates the address of both aggressor and victim memory cells, we extend the FP model by introducing the *Addressed Fault Primitive* concept.

Definition 4: An *Addressed Fault Primitive* (AFP) is an instantiation of a FP which explicit the involved addresses, and both the faulty and fault-free final memory state, reached by the memory, after applying the AFP. It can be formalized as:

$$AFP = (I , E_s , F_v , G_v) \quad (4)$$

where:

- $I = \{ (s)^{\#IC} \mid s \in C \}$ is the initial state, i.e., the value stored in the #IC involved cells, before applying the AFP. The first value correspond to the less significative bit (i.e. the memory cell with the lowest address);
- $E_s = \{ (op)^m \mid op \in X \}$ is the sequence of operations, performed on the aggressor cells, needed to sensitize the fault; each operation belong to the alphabet X, the set of all the possible memory operations. *m* is the number of operations needed to sensitize the fault;
- $F_v = \{ (f)^{\#IC} \mid f \in C \}$ is the logical value stored in the memory cells after applying Es (faulty state)
- $G_v = \{ (g)^{\#IC} \mid g \in C \}$ is the logical value stored in the memory cells after applying E_s on the fault-free memory (expected state).

The FP of the above example $< 0w_1 ; 0 / 1 / - >$ can be translated into AFP1 = (00, w^0_1, 11, 10) and AFP2 = (00, w^1_1, 11, 01), with a memory having $n = 2$ (i.e., two cells).

4. Linked Fault: Concept & Modeling

In some cases it is possible that the effect of a FFM influences another functional fault. If these faults share the same aggressor and/or victim cells, the FFMs are called *Linked*, otherwise they are called simple or un-linked and each fault is independent from the others. To understand the concept of linked faults we can consider, as an example, the Disturb Coupling Faults [14] described by the following two FPs:

$$FP_1 = < 0w_1 ; 0 / 1 / - >, \ FP_2 = < 0w_1 ; 1 / 0 / - > \quad (5)$$

A general case is represented in Figure 2, in which a *n* cells memory is affected by two FPs (FP1 and FP2) having different *a*-cells (a_1, a_2) and the same *v*-cell (v). The vertical arrow shows the address order of the memory (from the lowest memory address to the highest) in which *i*, *j* and *k* represent the address of a_1, a_2 and *v*, respectively. By first performing "$0w_1$" (FP1) on cell *i*, the *v*-cell *k* flips from 0 to 1; than performing "$0w_1$" (FP2) on cell *j*, the *v*-cell *k* changes its value again, from 1 to 0. The global result is that the fault effect is masked by the application of FP2, since FP2 has a fault effect (F) opposite to FP1. Looking at the example of Figure 2, we can derive a rigorous definition of a Linked Fault (LF):

Definition 5: Two FPs, FP1 = <S1/F1/R1> and FP2 = <S2/F2/R2>, are said to be *Linked,* and denoted by "FP1 $\rightarrow$ FP2", if both of the following conditions are satisfied:

- FP2 masks FP1, i.e., F2 = not (F1);
- The Sensitizing operation (S2) of FP2 is applied after S1, on either the a-cell or v-cell of FP1.

To detect linked faults (LFs), one must detect in isolation (i.e., without allowing the other FP to mask the fault) at least one of the FPs that compose the fault [6]. We extend the concept and the notation described in Definition 6 resorting to AFP (4) formalism.

Definition 6: Two AFPs, AFP1 = (I_1, E_{s1}, F_{v1}, G_{v1}) and AFP2= (I_2, E_{s2}, F_{v2}, G_{v2}) are said to be *Linked,* and denoted by "AFP1 $\rightarrow$ AFP2" if:

- $I_2 = F_{v1}$: the state reached by AFP1 is equal to initial state of AFP2;
- AFP2 masks AFP1 : $V(F_{v2})$ = NOT $[V(F_{v1})]$ where V(s) function extracts the victim cell from the memory state *s*

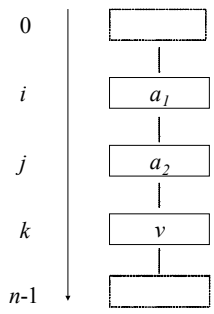

Figure 2. Example of Linked Fault

As an example the Linked Fault represented in (5), includes two AFPS obtained by FP1 and FP2:

- AFP1 = (000, w^0_1, 101, 100) (6)
- AFP2 = (101, w^1_1, 110, 111)

If we set the size of memory cell n = 3 (0,1,2), the a-cell is cell 0 in AFP1 and cell 1 in AFP2. Both AFP have v-cell equal 2. These two AFP satisfy the constraint of Definition 7:

- $I_2 = 101 = F_{v1}$
- $V(F_{v2}) = 0 = NOT\ [V(F_{v1}) = 1]$

Two AFP according to Definition 7 can model each Linked faults.

5. Fault Graph And Memory Model

The proposed memory fault simulator algorithm uses a memory model based on Finite State Machine (FSM). An n one-bit cells memory can be represented as a deterministic Mealy Automata, formally defined as:

$$M = (Q, X, Y, \delta, \lambda) \quad (7)$$

where:

- $Q = \{ (0| 1| -)^n \}$ is the set of possible *memory states*;
- X is the input alphabet defined in (2);
- $Y = \{0,1,-\}$ is the *output alphabet*, composed of the possible values read as a result of a read operation; '–' denotes the value obtained when a write operation is performed;
- $\delta = Q \times X \rightarrow Q$ is the *state transition function;*
- $\lambda = Q \times X \rightarrow Y$ is the *output function.*

The memory model defined in (7) can be represented as a labeled direct graph

$$G = \{V, E\} \quad (8)$$

where:

- V is the set of vertices, each vertex representing one of the possible states of the memory; $|V| = 2^n$,
- E is the set of edges, each edge representing one of the possible memory operations that cause the transition from a vertex u to a vertex v; the k^{th} label associated with the k^{th} edge has the following representation:

$$Label_k = x / d \quad (9)$$

where:

- $x \in X$ is a memory operation
- $d = \lambda(v,x)$, $d \in Y$ is the output value obtained when performing the operation x when the memory is in the state v.

As an example, Figure 3 shows the model of a 2 bit memory, conventionally named G_0 in the sequel. In G_0, the letters i and j are used to identify the first and the second cell, respectively. Hereinafter, we shall assume $i < j$. According to Definition 7 each Linked Fault is covered by a sequence of two Address Fault Primitives AFP1 and AFP2, each AFP can be modeled on G_0 by a single additional edge (*faulty edge*) [2] The couple of Linked AFPs is represented on the graph by adding two extra faulty edges as shown in Figure 4. In the graph representation, the memory state reached by AFP1 (F_{v1}) is equal to I_2 (initial state of AF2) in order to satisfy Definition 7.

The graph including the faulty edges is named *Pattern Graph* (PG) and is defined as:

$$PG = \{V_p, E_p \cup F_p\} \quad (10)$$

where each vertex $v \in V_p$ is associated to a memory state, E_p is the set of edges modeling the fault free memory, and F_p is the set of faulty edges. If n is the number of cells composing the target memory the number of nodes composing the pattern graph is $|V_p| = 2^n$. The cardinality of PG vertices is $2^{\max(\#f\text{-}cells_i)}$ with $0 \le i \le \#FP$, where #FP represents the number of FPs in the target fault list [2]. Moreover, the user can specify the size (i.e. the number of cell) of the memory by the *Memory Model file* (see Section 2), it is therefore possible model a real memory.

Definition 7: given f_l , $f_k \in F_p$, f_l masks f_k if and only if $V(F_{vk}) = V(I_l)$, where f_l is incident from vertex I_l is the and f_k is incident in vertex F_{vk} . V(s) is defined in Definition 6.

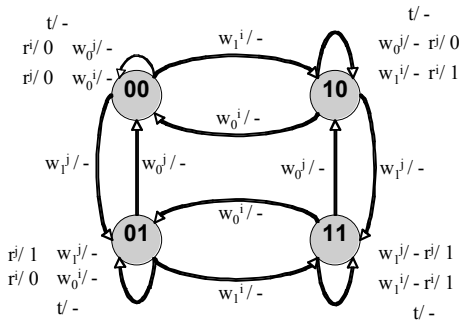

Figure 3. Fault Free Memory Model G_0

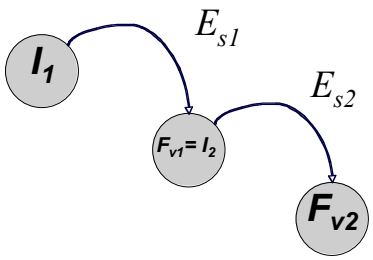

Figure 4. Linked Test Pattern Representation

As an example Disturb Coupling Fault linked to Disturb Coupling Fault [14] is modeled as two FPs:

$$< 0w_1 ; 0 / 1 / - > \rightarrow < 1w_0 ; 1 / 0 / - > \qquad (11)$$

Expressing the FPs in terms of AFPs we obtain:

$$(00, w^i_1, 11, 10) \rightarrow (11, w^1_0, 00, 01) \qquad (12)$$

The PG (named PG_{CF} in the sequel) modeling LF (12) is shown in Figure 5 where the bold edges represent the additional *faulty edges* of (14).

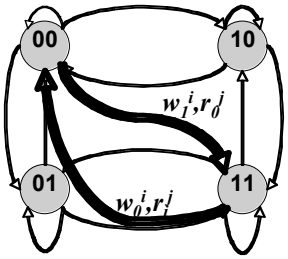

Figure 5. Linked CF Pattern Graph (PG_{CF})

6. Simulator Engine

The simulator algorithm is based on a *functional model*, represented as a Finite State Machine (see Section 5) where each Functional Faults is represented as en edge.

After generation of the memory model from the *fault model* and *Memory model* file, we inject one fault a time in the memory model (in case of linked faults we inject two faults a time) then the algorithm reads from the *Test Sequence* file the operation to be applied on the memory

The user can specify the *data-background* (DB), defined as the pattern of ones and zeros as seen in an array of memory cells. The most common types of data-backgrounds are: Solid (s), Checkerboard (c), Column Stripe (cs), and Row Stripe (rs) (Figure 6 shows an example of DB in case of 3x3 memory cells array).

The algorithm starts from a vertex depending on the DB selected by the user (i.e. in case of solid-0 DB and two cell memory model, the initial state will be '00'). Then if the memory operation to be applied traverses one Faulty Edge it means that the related fault has been sensitized. A fault is detected if a memory read operation returns a different value from the expedect one.

The simulation algorithm complexity depends from three factors:

- Number of memory cell: *n*
- Number of fault: #F
- Complexity of the test algorithm expressed in terms of memory operation: #MO

From these three factors we can calculate the complexity as equal to *O*(#F x *n* x #MO)

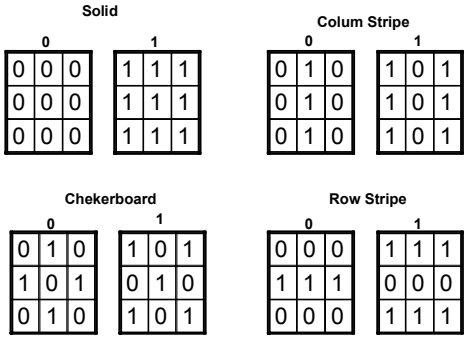

Figure 6. Data Background

7. Experimental Results

This section reports some experimental results obtained by applying the proposed simulation algorithm to different fault lists and different test algorithm. The algorithm has been implemented in about 10000 lines of C++ code, compiled with gcc compiler. All the experiments are performed on an *ASUS, AMD 1500Mhz* based Laptop with 512 MB of RAM.

We simulate and compare several March test on the same fault list and memory model. The fault lists include the set of realistic linked faults presented in [6] for a total of 552 fault primitives:

The adopted memory model consider the minimum number of required memory cell, since we are dealing with linked faults no more than three cells are required (n = 3). Eventually we consider the well known march test targeting the linked faults space (A, B, LR, LA, [3], March SL, March MSL, March AB, March RAW).

Table 1 summarizes the simulation results; each row represents a March test ordered by complexity (column 4).

Then we show the percentage of coverage in the different fault lists and eventually (last column) the fault coverage in the full set if linked faults. The underlined cells in table 1, point up the March tests that reach the 100% of coverage w.r.t. the considered Fault List. Experimental results show that March RAW [11] originally designed for covering dynamic un-linked faults only also covers the set of realistic static linked faults. The CPU time is negligible in fact it takes in the worst case 2.34 second.

8. Conclusions

This paper proposed a memory fault simulator targeting the entire set of linked memory faults. The proposed tool addresses the problem of the efficient and fast validating of memory test algorithm w.r.t. complex fault models or new memory structures

We also provide a rigorous description in terms of fault model and memory representation in order to better understand the simulation algorithm and the concept of linked faults in random access memory. Moreover we give the detailed percentage of linked fault coverage for each known march test.

9. References

[1] International Technology Roadmap for Semiconductors, *"International technology roadmap for semiconductors 2004 Update"*, http://public.itrs.net/Home.htm, 2004.

[2] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, P. Prinetto, "*Automatic March Tests Generation for Static and Dynamic Faults in SRAMs*", ETS 2005, 10th IEEE European Test Symposium, 2005.

[3] S. M. Al-Harbi, S. K. Gupta, *"Generating Complete and Optimal March Tests for Linked Faults in Memories",* VTS 2003, 21th IEEEVLSI Test Symposium, 2003, pp. 254 -261

[4] A. J. van de Goor, B. Smit, "*Generating March Tests Automatically*", ITC 1994, IEEE International Test Conference, 1994, pp.870-877, 1994

[5] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, P. Prinetto, "*Automatic March Tests Generations for Static Linked Faults in SRAMs*", DATE 2006, IEEE Design Automation and Test in Europe 2006

[6] S. Hamdioui, Z. Al-Ars, A. J. van de Goor, M. Rodgers, *"Linked Faults in Random Access Memories Concept Fault Models Test Algorithms and Industrial Results"*, IEEE Transaction on Computer-Aided Design, Volume: 23, Issue: 5, May 2004, pp. 737-757.

[7] A. J. van de Goor, "*Testing Semiconductor Memories: Theory and Practice"*, John Wiley & Sons, Chichester, England, 1991.

[8] S. Demidenko, A. van de Goor, S. Henderson, P. Knoppers, "*Simulation and Development of Short Transparent Tests for RAM*", IEEE 10th Asian Test Symposium (ATS 2001), Kyoto (J), November 2001

[9] C. Wu, C. Huang, C. Wu, "*RAMSES: a fast memory fault simulator*", International Symposium on Defect and Fault Tolerance in VLSI Systems, 1999, Page(s): 165 -173

[10] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, *"Specification and design of a new memory fault simulator"*, ATS 2002, 11th IEEE Asian Test Symposium, 2002.pp. 92 – 97.

[11] S. Hamdioui, Z. Al-Ars, A. J. van de Goor, "*Testing Static and Dynamic Faults in Random Access Memories*", VTS 2002, 20th IEEE VLSI Test Symposium, 2002, pp.

[12] A. J. van de Goor, G.N. Gayadadjiev, V.N. Yarmolik, V.G. Mikitjuk, *"March LA: A Test for Linked Memory Faults",* ED&TC 1997, Proc. European Design and Test Conference, 1997, pp. 167.

[13] A. J. van de Goor, G.N. Gayadadjiev, V.N. Yarmolik, V.G. Mikitjuk, *"March LR: A Test for Realistic Linked Faults",* VTS 1996, 16th IEEE VLSI Test Symposium, 1996, pp. 272-280.

[14] A. J. van de Goor, Z. Al-Ars, *"Functional Memory Faults: A Formal Notation and a Taxonomy"*, VTS 2000, 18th IEEE VLSI Test Symposium, 2000, pp. 281-289.

[15] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, P. Prinetto, "*March AB, March AB1: New March Tests for Unlinked Dynamic Memory Faults*", *ITC 2005, IEEE International Test Conference*, 2005.

[16] G. Harutunyan, V.A. Vardanian, Y. Zorian, "*Minimal March Test Algorithm for Detection of Linked Static Faults in Random ùAccess Memories*", DATE 2006, IEEE Design Automation and Test in Europe 2006

Table 1. Simulation results for both single and two/three -cells LF

MT	*Rif*	*CPU time (s)*	*O(n)*	*Single Cell*	*(Two/Three)-cells*			*All*
					$LF2_{aa}$	$LF2_{av}$	$LF2_{va}$	
LR	[12]	0.3	14*n*	75%	82%	75%	80%	80%
A	[7]	0.2	15*n*	66%	75%	60%	73%	69%
B	[7]	0.43	17*n*	75%	70%	64%	73%	70%
LA	[13]	1.02	22n	83%	87%	83%	86%	86%
AB	[15]	0.97	22n	100%	100%	100%	100%	100%
MSL	[16]	0.99	23n	100%	100%	100%	100%	100%
RAW	[11]	1.12	26n	100%	100%	100%	100%	100%
ABL	[5]	1.34	37*n*	100%	100%	100%	100%	100%
SL	[6]	2.01	41*n*	100%	100%	100%	100%	100%
-	[3]	2.34	43*n*	83%	84%	83%	86%	84%